Synthesis and Transformation of Logic Programs through Constructive, Inductive Proof

Synthesis and Transformation of Logic Programs in the *Whelk* Proof Development System

Geraint A Wiggins
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, Scotland
geraint@ed.ac.uk

Abstract

I present the *Whelk* proof development system, a tool for the synthesis and transformation of logic programs. *Whelk* is based on ideas used in the "Proofs-as-Programs" literature [9, 1], which enable the *extraction* of logic programs from proofs, in a similar way to that of [10]. Using *Whelk*, we can synthesise *pure logic programs* [4] which may be easily translated into "real" logic programming languages such as Prolog and Gödel. Delay declarations to prevent floundering and unbounded recursion may also be generated automatically. In the longer term, *Whelk* will form a substrate for the CLaM proof planner [7, 6, 8, 5], which will allow automatic generation of programs from first order specifications.

In this paper, I present a subset of the refinement rules used in *Whelk*. I show how a proof of correctness of both the rules and the synthesised program should be carried out. I give a detailed example of a synthesis/transformation proof.

1 Introduction

In this paper, I present and explain the *Whelk* proof development system. *Whelk* is intended for the synthesis and/or transformation of logic programs as I will explain below, though it is usable for the development of more general proofs. *Whelk* is designed for use with the CLaM proof planner [7, 6, 8, 5], which allows the automation of the proof process; currently, CLaM is orientated towards proofs by induction.

The rest of the paper takes the following form. Section 2 begins by outlining the notion of *proofs as programs*, summarising the problems raised and solutions proposed in [4], which arise when the idea is applied to *logic* program synthesis, rather than that of *functional* programs, which has been past practice. Section 3 summarises the operation of the *Whelk* system, and lists and justifies the proof rules used in the example of Section 4. Finally, Section 5 draws conclusions and outlines the next steps in the work presented here.

2 Background

2.1 Proofs as (Functional) Programs and Type Theory

Proofs-as-Programs is an existing technique for program synthesis [9]. The basic notion behind it is that mathematical proofs of certain theorems can contain the computational information required to construct a program. In particular, if we

prove a conjecture of the form

$$\vdash \forall \vec{i}.\exists o.S(\vec{i}, o)$$

where $\vec{i}$ is a (possibly empty) vector of inputs, o is an output, and S is the specification of a program, it is sometimes possible to derive that program from the proof. I will refer to this kind of conjecture as a *synthesis conjecture*.

In order to ensure that we can derive a program from the proof, and to ensure that the program is then executable, we must place restrictions on the proof system we use. The most straightforward way to achieve the necessary restriction is to require that the logic used for the proof be *constructive*. The upshot of this is that any proof of our existential goal, above, will involve showing not merely that the output value o exists, but that it can be constructed. This is shown by constructing it in terms of any other values, functions, and relations in S, and their definitions. Since the proof shows how to construct our output for all input, we can then derive a program which will construct it for any given input value.

This construction process begs an important question: how can we know that the program we construct is correct with respect to the proof? In the literature on functional proofs-as-programs (as opposed to proofs-as-relational-programs, which I discuss below), there is a standard technique for doing this. We cast our specification conjecture and its proof in a Constructive Type Theory; once we have done so, we can derive functions, expressed in terms of the λ-calculus, as our programs, and know they are correct by virtue of the *Curry-Howard Isomorphism* [14], a one-to-one relationship between the notions of implication and function application [16, 17].

2.2 Proofs as Relational Programs

The main problem with the application of the type-theoretic proofs-as-programs technique to the synthesis of logic programs is that it synthesises functions — if we were to synthesise logic programs which were strictly functional, we would be throwing away the main advantages of logic programs: multiple outputs, partial programs, the concept of failure, and so on. So we would like to adapt the technique.

In [4], we propose an adaptation of the proofs-as-programs technique in which we view relations (*qua* logic programs called in the *all-ground mode*) as functions on to the type *boole* (containing just *true* and $false$). We then prove a synthesis conjecture of the approximate form

$$\vdash \forall \vec{i}.\exists \mathcal{B}.\mathcal{S}(\vec{i}) \leftrightarrow \mathcal{B}$$

where $\mathcal{B}$ is a variable of type *boole*, $\vec{i}$ is a (possibly empty) vector of arguments and S is as before. This technique allows us to overcome the limitation of [10], which can only synthesise functional relations.

The exact form of the specification conjecture is largely a matter of taste — it can be specified as above, where the boolean variable is second-order, or in other ways which keep the logic first-order; we discuss these possibilities in [22]. The option I have chosen is to use a first order typed constructive logic with a new operator,

$$\partial : formula \mapsto formula$$

read as "it is decidable whether...". The meaning of ∂ is defined thus:

$$\vdash \forall \vec{a}.\partial\, S(\vec{a}) \quad \textit{iff} \quad \vdash \exists P.\forall \vec{a}.S(\vec{a}) \leftrightarrow P(\vec{a}) \text{ and } P, \text{ a relation, is decidable}$$

where S is a formula specifying a program. Proving decidability is equivalent to proving the existence of the boolean $\mathcal{B}$ in the initial approach. However, it has the advantage that the higher-order component of the specification, $\mathcal{B}$, is hidden away in the definition of the logic, and we are left with a purely first order specification.

Given this operator, branches of the proof corresponding with any parts of a logic program search tree can be elaborated normally, regardless of their eventual success or failure, and then related with *true* or $false$, in a synthesised program, as appropriate. This will become clearer in the exposition of the *Whelk* system in Section 3, and in the worked example in Section 4.

Now, since I am not using type theory (which is for various practical reasons not ideally suited to this work), I must also motivate the construction steps associated with the proof rules — I will do this in Section 3.3.

2.3 Justification

It is necessary to justify the usefulness of this technique as opposed to the many existing techniques for program synthesis and transformation in the logic programming world. It is regrettably the case that the proofs necessary to synthesise programs are often long-winded and difficult; it seems likely that direct writing of a program will be a much quicker and less laborious approach than this. However, I suggest that the technique is very much worthwhile, for the following reasons.

1. It involves working with logical specifications with no procedural content at all. Thus, is is much closer to the original intent of Kowalski's equation [15] than working with programs which have a procedural interpretation.

2. Working with specifications in terms of equivalence means that there is only one notion of program equivalence, which simplifies matters greatly.

3. The proofs associated with a particular theorem contain much more information than a logic program usually does. This information is available for use in connection with program synthesis and/or execution, once the proof has been carried out. For example, in [21], I explain how information about applications of induction can be used to generate delay declarations.

4. In this technique, automation of the synthesis process becomes for the most part equivalent to that of the proof process. Existing work (*eg* [7, 6, 8]) can therefore be used to generate programs automatically; thus, long-windedness of the proofs no longer matters.

5. Given 3 and 4, we can expect to adapt the existing work to use the information in the proof to produce *good* programs as part of the automated process more easily than in techniques working by program transformation alone.

6. Once the proof and extraction systems are shown to be correct, we know that a complete proof will give rise to a correct program. Thus, no correctness proofs are needed for the synthesised programs.

7. We can use the technique to reproduce the behaviour of other techniques — for example, in [20], I show how a prototype of the current technique can be used to reproduce the results of [2]. Given 6 above, this means that such techniques need no longer be proven correct — if they are implemented in the synthesis logic, then they must be so, *a priori*, given that their notion of program equivalence is the same as *Whelk*'s.

3 The *Whelk* System

3.1 Introduction

Whelk is a Gentzen Sequent Calculus proof development system, based on the Oyster system of [7], the Martin-Löf-based Type Theory of that earlier system having been replaced by a first order typed constructive logic with equality. *Whelk* is designed and implemented as a substrate of the CLaM proof planner, as explained in Section 5.

In this section, I will state the refinement and construction rules necessary for the example of Section 4, which constitute a good cross-section of the rules of the full *Whelk* system.

3.2 The Logic

The proof refinement system I use here is based on a sequent calculus, LJ, of [11]. LJ has been enhanced for our purposes by the addition of the ∂ operator and refinement rules for each combination of ∂ with the other operators, and with types and equality. It would take too much space here to state the rules for the whole system, so I will limit myself to those necessary for the example proof in Section 4. These rules are laid out in the following sections.

Notation is as follows: upper case Greek letters (Γ, Δ) denote sequences of formulæ and program fragments; lower case Greek letters (τ) denote types and program fragments; upper case Roman letters (A, B, C) denote formulæ; and lower case Roman letters denote variables (x, v_i) or terms (t). $\{\}$ denotes contradiction. $A\langle t/x\rangle$ means "A with all free occurrences of x replaced by t". Finally, we have the usual connectives, $\forall$, $\exists$, $\wedge$, $\vee$, $\leftarrow$, $\rightarrow$, $\neg$, with the addition of $\leftrightarrow$; we have typed $=$, and we have $:$ denoting type membership.

In all the rules, where a term substitution occurs, the type of the replacement term must be consistent with that of the term being replaced.

Note also that there must (for the moment) be no more than one occurrence of ∂ in a goal. Multiple occurrences would mean that we were synthesising meta-programs, which I defer for a future document.

3.3 Extracting Pure Logic Programs

We also need to supply a set of construction rules, in correspondence with the proof rules, to allow our program to be built. This way, the program extraction process can take place as a side effect of elaborating the proof, and then the program can simply be read off afterwards.

The language I will use for my extracted programs is that of the *pure logic program* used in [4] and [22]. It is essentially the same as the specification language used above, but functions other than constructors are not allowed, and the ∂ operator does not appear. Also, the boolean terms *true* and $false$ are replaced by the predicates of the same name. There is a mapping, the *interpretation*, between the two logics, which I will now label as $\mathcal{L}_{\mathcal{S}}$ and $\mathcal{L}_{\mathcal{P}}$ – the *specification logic* and the *program logic*, respectively. Note that, in [22], these were called $\mathcal{L}_{\mathcal{E}}$ and $\mathcal{L}_{\mathcal{I}}$, the "external" and "internal" logics respectively. I no longer use this terminology, which can be misleading. The correspondent of an $\mathcal{L}_{\mathcal{S}}$-formula A in $\mathcal{L}_{\mathcal{P}}$ under the interpretation, is denoted by A^*; the only detail necessary here is that the connectives of $\mathcal{L}_{\mathcal{S}}$ map to connectives of the same name and meaning in $\mathcal{L}_{\mathcal{P}}$ under *. See [22] for more details.

The behaviour we require of our system is that, given a specification conjecture

$$\vdash \forall \vec{a}.\partial\, S(\vec{a})$$

we synthesise a pure logic program $P(\vec{a})$ such that

$$\vdash \forall \vec{a}^*.S(\vec{a})^* \leftrightarrow P(\vec{a})$$

so that P is effectively the "witness" for the decidability of the specification S – *ie* something which actually does the deciding.

For our purposes here, it is enough to say that a pure logic program, like that defining P, consists of a head and a body, connected biconditionally, and that the body is an arbitrary formula containing quantification, and maybe free occurrences of variables appearing in the head, all variables being typed. It may also contain explicit *true* and $false$ predicates, so the natural expression of a pure Prolog program in these terms is its completion. For example, the Prolog `member/2` predicate

```
member( X, [X|_] ).
member( X, [_|Y] ) :- member( X, Y ).
```

could be expressed, for lists of natural numbers, as the following pure logic program:

$$\begin{aligned} &member(x{:}nat, y{:}list(nat)) \leftrightarrow \\ &\quad y = [\,] \wedge false \vee \\ &\quad \exists v_0{:}nat.\exists v_1{:}list(nat).y = [v_0|v_1] \ \wedge (x = v_0 \wedge true \vee \\ &\qquad\qquad\qquad\qquad\qquad\qquad member(x{:}nat, v_1{:}list(nat))) \end{aligned}$$

To generate our desired program, we need a construction rule corresponding with each proof rule. First, I introduce notions of *synthesis proof* and *verification proof*.

Synthesis proof is that part of the proof of a specification conjecture which contributes directly to the extracted program. *Verification proof* is that part which shows the synthesis part to be correct, but does not actually constitute part of the synthesised algorithm itself. This distinction is also made in the functional proofs-as-programs literature — though, in Martin-Löf type theory, both the synthesis and the verification parts of the proof contribute to the extracted program, which is undesirable for our purposes here.

It is a desirable feature of the logic $\mathcal{L}_{\mathcal{S}}$ that it is possible and easy to distinguish syntactically between synthesis and verification parts of the proof: wherever there is a ∂ in one's conjecture, one is working on synthesis; elsewhere one is performing verification. This is reflected in the construction rules shown below.

The rule governing whether or not manipulation of a hypothesis contributes to the construction of a program is more subtle than that for conjectures. With a few exceptions, hypotheses in sequents are considered to be *true* and are therefore associated with the program fragment *true* by default. The only times when this is not the case are:

- when a disjunctive hypothesis is split into its disjuncts — each disjunct is then a constraint on the environments of a separate subconjecture;
- when an induction step is used — when the program fragment associated with the induction hypothesis is a recursive call to the synthesised predicate; and
- when a hypothesis is *cut* in, or a lemma is appealed to — in which case the program fragment associated with the new hypothesis is a call to the program synthesised by its proof (or just *true* if it does not contain ∂).

All of these will be demonstrated in Section 4.

Notation is as follows: each sequent or hypothesis A must now be associated with a program fragment; these fragments are written as subscripts on the expressions, each one with an explicit fragment being placed in $[\![\ ]\!]$ for clarity. Where no value is given, if A is decidable by first-order unification (*eg* a function-free equality) the default is A^*; otherwise the value is a program fragment ϕ constructed by proof of the subconjecture

$$[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E}) \leftrightarrow \phi}$$

The body of the synthesised program fragment associated with each subconjecture is denoted by a greek letter (ϕ, ψ); the head, $P(\mathcal{E})$, is a predicate symbol, P, applied to an "environment" of variables, $\mathcal{E}$, which constitute the parameters of the predicate. $\mathcal{E}$ is written here as a list in $\langle ., nil \rangle$ notation. The program fragment associated with the dominating conjecture is then expressed in terms of those of its subconjectures.

The refinement rules given in Section 3.4 are in some cases annotated with rules for program construction. Those which have no such annotation simply do not contribute to the construction; they serve only to verify its correctness. Those rules marked † do not apply to formulæ containing ∂.

Finally, note one further non-standard feature of the logic. Any formula of form

$$\partial\, A$$

where A is a formula is necessarily true — it states that A is either true or false. It is nevertheless necessary to give refinement rules for the *provability* ($\vdash$) of conjectures of this kind, because of their contribution to the synthesised program (see below).

3.4 The Refinement and Construction Rules

3.4.1 Axiom

We need two forms of axiom rule, the second deriving anything from contradiction:

$$\mathit{axiom}\ \frac{}{[\![\Gamma, [\![A]\!]_{\phi}, \Delta \vdash A]\!]_{P(nil) \leftrightarrow \phi}} \qquad \mathit{axiom}\dagger\ \frac{}{\Gamma, \{\}, \Delta \vdash A}$$

3.4.2 $\exists$ Introduction

$$\exists\ \mathit{intro}\ \frac{\Gamma \vdash A\langle t/x \rangle}{\Gamma \vdash \exists x{:}\tau.A}$$

3.4.3 $\forall$ Introduction

We need two forms of $\forall$ introduction, distinguishing between $\forall$ outside and inside the scope of ∂. This is necessary because the first rule leads to the introduction of a parameter into $\mathcal{E}$ and the second does not.

$$\forall\ \mathit{intro}\ \frac{[\![\Gamma, x{:}\tau \vdash \partial\, A]\!]_{P(\mathcal{E}) \leftrightarrow \phi}}{[\![\Gamma \vdash \forall x{:}\tau.\partial\, A]\!]_{P((x:\tau).\mathcal{E}) \leftrightarrow \phi}} \qquad \forall\ \mathit{intro}\ \frac{[\![\Gamma, x{:}\tau \vdash \partial\, A]\!]_{P(\mathcal{E}) \leftrightarrow \phi}}{[\![\Gamma \vdash \partial\, (\forall x{:}\tau.A)]\!]_{P(\mathcal{E}) \leftrightarrow \phi}}$$

3.4.4 $\wedge$ Introduction

$$\wedge\ \mathit{intro}\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E}) \leftrightarrow \phi} \qquad [\![\Gamma \vdash \partial\, B]\!]_{P(\mathcal{E}) \leftrightarrow \psi}}{[\![\Gamma \vdash \partial\, (A \wedge B)]\!]_{P(\mathcal{E}) \leftrightarrow \phi \wedge \psi}}$$

3.4.5 $\vee$ Introduction

$$\vee\ \mathit{intro}\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E}) \leftrightarrow \phi} \qquad [\![\Gamma \vdash \partial\, B]\!]_{P(\mathcal{E}) \leftrightarrow \psi}}{[\![\Gamma \vdash \partial\, (A \vee B)]\!]_{P(\mathcal{E}) \leftrightarrow \phi \vee \psi}}$$

3.4.6 $\rightarrow$ Introduction

We need two forms of $\rightarrow$ introduction, distinguishing between $\rightarrow$ inside and outside the scope of ∂, as with $\forall$. Note that, in the construction rule, we must reject the rule of *ex falsio quod libet*, to avoid proof of anything from falsehood, which is undesirable in logic programming. Thus, that $\rightarrow$ in $\mathcal{L}_{\mathcal{P}}$ is more like (local cut) in Prolog than classical implication as in Gödel, though it does not have the same (or indeed any) procedural interpretation.

$$\rightarrow \mathit{intro}\ \frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \qquad \rightarrow \mathit{intro}\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\phi} \quad [\![\Gamma, A \vdash \partial\, B]\!]_{P(\mathcal{E})\leftrightarrow\psi}}{[\![\Gamma \vdash \partial\,(A \rightarrow B)]\!]_{P(\mathcal{E})\leftrightarrow\phi\rightarrow\psi}}$$

3.4.7 ∂ Introduction

$$\partial_{\,true}\ \mathit{intro}\ \frac{[\![\Gamma \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathit{nil})\leftrightarrow \mathit{true}}} \qquad \partial_{\,false}\ \mathit{intro}\ \frac{[\![\Gamma \vdash \neg A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathit{nil})\leftrightarrow \mathit{false}}}$$

3.4.8 $\forall$ Elimination

For this rule, we need all three possibilities: $\forall$ inside and outside ∂, and in expressions not containing ∂. $\mathcal{E}''$ is $\mathcal{E}'$ prepended to $(t{:}\tau).\mathit{nil}$.

$$\forall\ \mathit{elim}\ \frac{[\![\Gamma, [\![\partial \forall x{:}\tau.A]\!]_{P(\mathcal{E}')}, \Delta, [\![\partial\, A\langle t/x\rangle]\!]_{P(\mathcal{E}'')} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma, [\![\partial \forall x{:}\tau.A]\!]_{P(\mathcal{E}')}, \Delta \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

$$\forall\ \mathit{elim}\ \frac{[\![\Gamma, [\![\forall x{:}\tau.\partial\, A]\!]_{P(\mathcal{E}')}, \Delta, [\![\partial\, A\langle t/x\rangle]\!]_{P(\mathcal{E}'')} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma, [\![\forall x{:}\tau.\partial\, A]\!]_{P(\mathcal{E}')}, \Delta \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

$$\forall\ \mathit{elim}\dagger\ \frac{\Gamma, \forall x{:}\tau.A, A\langle t/x\rangle, \Delta \vdash B}{\Gamma, \forall x{:}\tau.A, \Delta \vdash B}$$

3.4.9 $\neg$ Elimination

$$\neg\ \mathit{elim}\dagger\ \frac{\Gamma, \neg A, \Delta \vdash A}{\Gamma, \neg A, \Delta \vdash \{\}}$$

3.4.10 $\vee$ Elimination

$$\vee\ \mathit{elim}\ \frac{\begin{array}{cc}[\![\Gamma, A, \Delta \vdash C]\!]_{P(\mathcal{E})\leftrightarrow\phi} & [\![\Gamma, B, \Delta \vdash C]\!]_{P(\mathcal{E})\leftrightarrow\psi} \\ [\![\Gamma, \Delta \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\alpha} & [\![\Gamma, \Delta \vdash \partial\, B]\!]_{P(\mathcal{E})\leftrightarrow\beta}\end{array}}{[\![\Gamma, A \vee B, \Delta \vdash C]\!]_{P(\mathcal{E})\leftrightarrow(\alpha\wedge\phi)\vee(\beta\wedge\psi)}}$$

Here, the constructions α and β ensure that the disjuncts in the eliminated disjunction correspond with executable program fragments.

3.4.11 Induction on Lists

This is the rule which will allow us to build recursive programs. (Of course, further induction rules exist in *Whelk* but we do not need them for the example given here.)

$$\mathit{induction}\ \frac{\begin{array}{c}[\![\Gamma, x{:}\mathit{list}(\tau), \Delta \vdash A\langle[\,]/x\rangle]\!]_{P'(\mathcal{E})\leftrightarrow\phi} \\ [\![\Gamma, x{:}\mathit{list}(\tau), \Delta, v_0{:}\tau, v_1{:}\mathit{list}(\tau), [\![A\langle v_1/x\rangle]\!]_{P'(\mathcal{E}')} \vdash A\langle[v_0|v_1]/x\rangle]\!]_{P'(\mathcal{E})\leftrightarrow\psi}\end{array}}{[\![\Gamma, x{:}\mathit{list}(\tau), \Delta \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\mathcal{P}'(\mathcal{E})}}$$

$\mathcal{E}'$ is the environment at the time of application of the rule, and P' is defined by

$$P'(\mathcal{E}') \leftrightarrow \S =_{\updownarrow\rangle\int\sqcup(\tau)} [\,] \wedge \phi \vee \exists \sqsubseteq_{\prime}{:}\tau.\exists \sqsubseteq_{\infty}{:}\tau.\S =_{\updownarrow\rangle\int\sqcup(\tau)} [\sqsubseteq_{\prime}|\sqsubseteq_{\infty}] \wedge \psi$$

Note that the program fragment associated with the induction hypothesis is a call to a procedure, as opposed to a procedure definition, such as are associated with sequents. This is always the case for hypotheses.

3.4.12 Substitution

We have two substitution rules, one under logical equivalence, which must be justified, and one for rewrites manipulating only the connectives in $\mathcal{L}_{\mathcal{S}}$.

$$sub\ \frac{\Gamma \vdash [\![B \leftrightarrow C]\!]_{P(\mathcal{E})\leftrightarrow\psi} \quad \Gamma \vdash [\![A\langle C/B\rangle]\!]_{P(\mathcal{E})\leftrightarrow\psi\wedge\phi}}{\Gamma \vdash [\![A]\!]_{P(\mathcal{E})\leftrightarrow\phi}} \qquad rewrite\ \frac{\Gamma \vdash [\![E]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{\Gamma \vdash [\![D]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

where $D \leftrightarrow E$, this being determined by the proof system.

In the *sub* rule, the contribution of ψ is to connect any variables instantiated in the proof of the substituted goal with those in the original.

3.4.13 New Hypotheses

$$lemma\ \frac{[\![\Gamma, [\![B]\!]_{P'(nil)} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

where P' is the program synthesised by the proof of the lemma, B.

3.5 Generating "Real" Logic Programs

Given the rules of Section 3.2, we can now perform a proof of the specification conjecture specified in 4.1. On completion of this proof, we will have a program which embodies the algorithm which we showed to exist during the proof process.

However, this program is a pure logic program, and not a program in a generally accessible programming language. Fortunately, transforming pure logic programs into Prolog and Gödel programs is easy, except for the issues of floundering and infinite looping mentioned below. A little trivial partial evaluation quickly removes all the failed branches of our program. Because our logic was constructive, we do not have disjunctive heads, so we immediately generate Horn Clauses.

For the rest of this paper, I shall focus on the Gödel language [13], because it gives us some features which are preferable, here, to those of Prolog. In particular, Gödel allows arbitrary formulæ in the bodies of its clauses, so there is no further work involved in unpacking the bodies of our pure logic programs. Much more importantly, Gödel admits explicit `DELAY` declarations (as found in NuProlog), which we can use to prevent floundering and infinite looping in our synthesised programs. [21] explains how the inductive structure of the synthesis proof encodes the information we need to generate these declarations without further analysis.

3.6 Correctness

3.6.1 Introduction

There is not space here to give the full correctness proof for this system. I will, however, sketch a proof for a few rules which will show how the proof is done. Correctness of the sequent calculus is presented in terms of an existing Gentzen Sequent Calculus, assumed correct *a priori*. Correctness of the synthesised programs with respect to the specifications is shown in terms of the required behaviour specified in Section 3.3. A full proof of the correctness of *Whelk* will be given elsewhere.

3.6.2 Correctness of the proof system

I start from $LJ_{\tau,=}$, a constructive logic based on LJ [11] with the addition of equality and sorts in the obvious way. Connectives have their usual constructive meanings.

Theorem 3.1 (Correctness of *Whelk* Verification Logic) *$V \vdash \phi$ iff $LJ_{\tau,=} \vdash \phi$ where V is the subset of Whelk rules not mentioning ∂ and ϕ does not mention ∂.*

Proof 3.1 V is identical with $LJ_{\tau,=}$ by definition. □

Theorem 3.2 (Correctness of Synthesis Logic) *The rules of Whelk* including ∂ *are correct with respect to the interpretation of* ∂ *given in Section 2.2.*

Proof 3.2 By definition, $\vdash \partial A$ iff $\vdash A \vee \neg A$. Rewrite any *Whelk* rule under this equivalence. The resulting rule is necessarily derivable from the rules of $LJ_{\tau,=}$. □

Theorem 3.3 (Correctness of induction rule) *The induction rule for finite lists preserves correctness with respect to the interpretation of* ∂ *given in Section 2.2.*

Proof 3.3 In the usual way. □

Theorem 3.4 (Correctness of *Whelk* Logic) *The rules of Whelk are correct with respect to the usual constructive interpretations of the standard connectives, and to that given for* ∂ *in Section 2.2.*

Proof 3.4 From Proofs 3.1, 3.2, and 3.3. □

3.7 Correctness of the synthesis system

Theorem 3.5 (Correctness of the synthesis system) *If* $P(\vec{a})$ *is a program synthesised by Whelk proof of a specification conjecture*

$$\vdash \forall \vec{a}.\partial\, S(\vec{a})$$

then

$$\vdash \forall \vec{a}^*.S(\vec{a})^* \leftrightarrow P(\vec{a})$$

Proof 3.5 By induction on the structure of proofs, building on base cases of the ∂ introduction rules, each construction rule being proven correct individually.

Base Case 1:
Consider

$$\partial_{true}\ intro\ \frac{[\![\Gamma \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash A]\!]_{P(nil)\leftrightarrow true}} \qquad \partial_{false}\ intro\ \frac{[\![\Gamma \vdash \neg A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash \partial\, A]\!]_{P(nil)\leftrightarrow false}}$$

Given a specification (sub)conjecture

$$\Gamma \vdash \partial\, S$$

suppose that $\Gamma \vdash S$. Then application of ∂_{true} introduction yields subconjecture

$$\Gamma \vdash S$$

which can be shown in $LJ_{\tau,=}$. The corresponding program is defined as a proposition

$$P \leftrightarrow true$$

Therefore, $\vdash S$ is true and $P \leftrightarrow true$, so $\vdash S^* \leftrightarrow P$. □

Base Case 2:
Alternatively, given a specification (sub)conjecture

$$\Gamma \vdash \partial\, S$$

such that $\Gamma \vdash \neg S$, application of ∂_{false} introduction yields subconjecture

$$\Gamma \vdash \neg S$$

which can be shown in LJ. The corresponding program is a proposition:

$$P \leftrightarrow false$$

Therefore, $\vdash S$ is false and $P \leftrightarrow false$, so $\vdash S^* \leftrightarrow P$. □

Step Case
Consider $\wedge$ introduction under ∂:

$$\wedge\ intro\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\phi} \quad [\![\Gamma \vdash \partial\, B]\!]_{P(\mathcal{E})\leftrightarrow\psi}}{[\![\Gamma \vdash \partial\,(A \wedge B)]\!]_{P(\mathcal{E})\leftrightarrow\phi\wedge\psi}}$$

Suppose we have a synthesised (sub)program

$$P \leftrightarrow \phi \wedge \psi$$

from a specification conjecture

$$\Gamma \vdash \partial\,(A \wedge B)$$

Then the required behaviour is

$$\Gamma^* \vdash (A \wedge B)^* \leftrightarrow P$$

which is equivalent to

$$\Gamma^* \vdash (A \wedge B)^* \leftrightarrow \phi \wedge \psi$$

It is known that

$$\Gamma^* \vdash A^* \leftrightarrow \phi$$

and

$$\Gamma^* \vdash B^* \leftrightarrow \psi$$

Therefore,

$$\Gamma^* \vdash (A^* \leftrightarrow \phi) \wedge (B^* \leftrightarrow \psi)$$

Therefore (recall that the meaning of connectives is preserved under *) it is not hard to show that

$$\Gamma^* \vdash (A \wedge B)^* \leftrightarrow \phi \wedge \psi$$

The correctness of the other construction rules is proven in the same way. □

4 Example: `subset/2`

4.1 A Simple Synthesis Conjecture

For the purposes of example here, I will use a conjecture which specifies the `subset/2` predicate using lists as a representation for sets — that is, the predicate which succeeds when all the members of the list given as its first argument are members of that given as its second. The specification in *Whelk* looks like this:

$$\vdash \forall x{:}list(nat).\forall y{:}list(nat).\partial\,(\forall z{:}nat.z \in x \rightarrow z \in y)$$

where : denotes type membership, $\forall$ and $\exists$ denote the usual quantification over types, $\rightarrow$ denotes the usual implication, and $\in$ is defined by the following lemmas (which are equivalent to the completion of the familiar `member/2` predicate). Lists are denoted with the Prolog/Gödel notation.

$$\vdash \forall x{:}nat.\neg x \in [\,] \tag{1}$$

$$\vdash \forall x{:}nat.\forall h{:}nat.\forall t{:}list(nat).x \in [h|t] \leftrightarrow x = h \vee x \in t \tag{2}$$

We will also use an axiom about the decidability of equality in the natural numbers:

$$\vdash \forall x{:}nat.\forall y{:}nat.x = y \vee \neg x = y \tag{3}$$

The proof proceeds by primitive induction on lists, first on x and then on y. Note, though, that there is no reason in principle why more powerful forms of induction should not be used. For example, insertion sort and quicksort may be derived from the same specification, the former by primitive induction, the latter by course-of-values or transfinite induction [18].

Necessarily, I have skipped some steps in this proof, because the proof is far too long to present in full here. However, I have tried to focus on the points which are most relevant to the synthesis issues discussed in this paper. Note, in particular, that the proof is presented in refinement style, with the rules being applied "backwards", and that I have omitted unchanging hypotheses unless they are used in the current proof step. The finished program, after conversion to Gödel, looks like this:

```
MODULE Subset.
IMPORT Lists.
IMPORT Numbers.

PREDICATE Subset :  List( Number ) * List( Number ).
Subset(y,z) <- Subset_b(y,z).

PREDICATE Subset_b :  List( Number ) * List( Number ) * Number * Number.
Subset_b(y,z) <-
    (y = [] \/
     SOME [v1] SOME [v0] ( y=[v0|v1] &
                           SOME [x] (x=v0 -> Subset_dy(z,x)) &
                                     Subset_b(v1,z)))).

PREDICATE Subset_dy :  List( Number ) * Number.
Subset_dy(z,x) <-
    SOME [v3] SOME [v2] (z=[v2|v3] &
                         (x=v2 \/ Subset_dy(v3,x))).
```

4.2 The Proof

We start with:

$$\vdash \forall x{:}list(nat).\forall y{:}list(nat).\partial\,(\forall z{:}nat.z \in x \rightarrow z \in y)$$

First, we introduce x, and apply primitive induction on lists. This gives us two subgoals (note the program fragment attached to the induction hypothesis in the step case — I use σ to denote the name of the synthesised predicate).

Base Case:

$$\begin{array}{l} x{:}list(nat) \\ \vdash \forall y{:}list(nat).\partial\,(\forall z{:}nat.z \in [\,] \rightarrow z \in y) \end{array}$$

Step Case:

$$\begin{array}{l} x{:}list(nat) \\ v_0{:}nat \\ v_1{:}list(nat) \\ [\![\forall y{:}list(nat).\partial\,(\forall z{:}nat.z \in v_1 \rightarrow z \in y)]\!]_{\sigma'((x{:}list(nat)).nil)} \\ \vdash \forall y{:}list(nat).\partial\,(\forall z{:}nat.z \in [v_0|v_1] \rightarrow z \in y) \end{array}$$

At this stage, the synthesised program looks like this:

$$\begin{array}{lcl}\sigma((x{:}list(nat)).\eta) & \leftrightarrow & \sigma'((x{:}list(nat)).\eta)\\ \sigma'((x{:}list(nat)).\eta]) & \leftrightarrow & (x = [\,] \wedge \phi) \vee \\ & & \exists v_0{:}nat.\exists v_1{:}list(nat).x = [v_0|v_1] \wedge \psi\end{array}$$

where η, ϕ and ψ are uninstantiated meta-variables. Comparing this with the Gödel definitions of `Subset/2` and `Subset_b/3` in Section 4.1 will yield a clearer understanding of which parts of the proof give rise to which program fragments.

I will now follow through the base case of the proof. We proceed by introducing the remaining universal quantifiers, and then by ∂_{true} introduction:

$$\begin{array}{l}y{:}list(nat)\\ z{:}nat\\ \vdash z \in [\,] \rightarrow z \in y\end{array}$$

The synthesised program now looks like this:

$$\begin{array}{lcl}\sigma((x{:}list(nat)).(y{:}list(nat)).nil) & \leftrightarrow & \sigma'((x{:}list(nat)).(y{:}list(nat)).nil)\\ \sigma'((x{:}list(nat)).(y{:}list(nat)).nil) & \leftrightarrow & (x = [\,] \wedge true) \vee \\ & & \exists v_0{:}nat.\exists v_1{:}list(nat).x = [v_0|v_1] \wedge \psi\end{array}$$

and we are left with a verification conjecture in standard $LJ_{=,\tau}$ which is trivially proven using definition (1).

The step case is harder. First, use the *sub* $\leftrightarrow$ rule to unfold the specification according to definition (2). Then introduce the universal quantifier of y and rewrite under propositional equivalence to get:

$$\begin{array}{l}y{:}list(nat)\\ \vdash \partial\,(\forall z{:}nat.(z = v_0 \rightarrow z \in y) \wedge \forall z{:}nat.(z \in v_1 \rightarrow z \in y))\end{array}$$

The rewriting can be performed automatically, via the *rippling* paradigm of [3, 8]. The only step so far in the step case affecting the synthesised program is the $\forall$ introduction: rewrites maintain logical equivalence, and so do not change the program.

Next, we introduce $\wedge$ under ∂ to give two subconjectures:

$$\vdash \partial\,(\forall z{:}nat.z = v_0 \rightarrow z \in y) \tag{4}$$

$$\vdash \partial\,(\forall z{:}nat.z \in v_1 \rightarrow z \in y) \tag{5}$$

The proof of (4) runs as follows. After introducing $\forall$ and $\rightarrow$, we use axiom (3) to decide on the equality of z with v_0, which leaves us with the program

$$\begin{array}{l}\sigma(x{:}list(nat).y{:}list(nat).nil) \leftrightarrow \sigma'(x{:}list(nat).y{:}list(nat).nil)\\ \sigma'(x{:}list(nat).y{:}list(nat).nil) \leftrightarrow x = [\,] \wedge true \vee \\ \qquad\qquad \exists v_0{:}nat.\exists v_1{:}list(nat).x = [v_0|v_1] \wedge \\ \qquad\qquad\quad ((x = v_0 \wedge true \vee \neg x = v_0 \wedge false) \rightarrow \phi) \wedge \psi\end{array}$$

where ϕ and ψ correspond with the right hand branches from the $\rightarrow$ and $\wedge$ introductions, above, respectively.

The sequent corresponding with ϕ is

$$\begin{array}{l}x = v_0\\ \vdash \partial\, z \in y\end{array}$$

This is proven by induction, using the definition of $\in$. It yields the `member/2` predicate (`Subset_dy/2` in Section 4.1).

Finally, (5) leads to introduction of the program fragment associated with the induction hypothesis, as follows. Recall that the goal is:

$$[\![\forall y{:}list(nat).\partial(\forall z{:}nat.z \in v_1 \rightarrow z \in y)]\!]_{\sigma'((x{:}list(nat)).nil)}$$
$$y{:}list(nat)$$
$$\vdash \partial(\forall z{:}nat.z \in v_1 \rightarrow z \in y)$$

We eliminate y on the induction hypothesis to yield the sequent:

$$[\![\forall y{:}list(nat).\partial(\forall z{:}nat.z \in v_1 \rightarrow z \in y)]\!]_{\sigma'((x{:}list(nat)).nil)}$$
$$y{:}list(nat)$$
$$[\![\partial(\forall z{:}nat.z \in v_1 \rightarrow z \in y)]\!]_{\sigma'((x{:}list(nat)).(y{:}list(nat)).nil)}$$
$$\vdash \partial(\forall z{:}nat.z \in v_1 \rightarrow z \in y)$$

The proof is then completed by application of the axiom rule.

We now have the following pure logic program:

$$\sigma(x{:}list(nat).y{:}list(nat).nil) \leftrightarrow \sigma'(x{:}list(nat).y{:}list(nat).nil)$$
$$\sigma'(x{:}list(nat).y{:}list(nat).nil) \leftrightarrow$$
$$x = [\,] \wedge true \vee$$
$$\exists v_0{:}nat.\exists v_1{:}list(nat).x = [v_0|v_1] \wedge$$
$$((x = v_0 \wedge true \vee \neg x = v_0 \wedge false) \rightarrow$$
$$\sigma''(y{:}list(nat).x{:}nat.nil) \wedge$$
$$\sigma'(v_1{:}list(nat).y{:}list(nat).nil)$$
$$\sigma''(y{:}list(nat).x{:}nat.nil) \leftrightarrow$$
$$y = [\,] \wedge false \vee$$
$$\exists v_2{:}nat.\exists v_3{:}list(nat).y = [v_2|v_3] \wedge$$
$$((x = v2 \wedge true \vee \neg x = v2 \wedge false) \vee$$
$$\sigma''(v_3{:}list(nat).x{:}nat.nil)$$

This then translates, trivially, into the Gödel program of Section 4.1.

5 Conclusion and Future Work

In this paper, I have demonstrated the theory behind the *Whelk* program synthesis and transformation system. I have outlined a proof of correctness for the system, and I have show how it can be used to develop a simple program.

The implications of this are as follows. We now have a proof system which will allow us to synthesise programs from logic specifications. Because the proof system and associated synthesis system is known to be correct, programs synthesised by it are also correct (with respect to the specification!) *a priori*. The approach improves over similar existing approaches (*eg* [10]) because it generates true logic programs with non-determinism, rather than only functional predicates.

Because there is strong connection between the steps taken in a proof (in particular between the choice of induction scheme and the resulting algorithm), we can exercise considerable control over the program we eventually obtain. Thus, we are in a position to use proof steps which we know will lead to efficient programs. This, however, is subject to certain controls in the object-level logic to ensure that *good* programs are produced — one of which requires, for example, that negation be partially evaluated as far as possible in the synthesised program, thus reducing (or usually removing) the problem of floundering [21]. In similar vein, `DELAY` declarations can be generated easily and automatically, using the inductive structure of the proof.

Subject to these desirable restrictions, the close connection between the proof rules and those for construction means that we can in principle implement other

techniques in our system. For example, one technique which has already been reconstructed in this framework is that of Compiling Control [2]. Another likely candidate for reconstruction is the block fold/unfolding work of [19]. A noteworthy point is that *Whelk* will provide a platform on which these techniques and others may not merely by developed and tested, but also combined in new and useful ways.

The next task required is to finish the implementation of the system and thence to being able to use the CLaM proof planner to generate programs automatically. This will require the construction of new meta-level encodings of proof strategies for producing not only proofs, but proofs which correspond with *efficient* programs, as in [12]. It is known that the *rippling* paradigm [5] can usually reduce the search tree for a proof to a linear path; the search heuristics are not, however, motivated towards program synthesis, and will usually produce the shortest proof, rather than the one corresponding with the most efficient program. This, then, will be the main focus of the forthcoming work with *Whelk*.

6 Acknowledgements

This work was carried out as part of ESPRIT Basic Research Action #3012 ("Computational Logic"). I am grateful to my colleagues in that project (especially Danny De Schreye, Jonathan Lever, and Torbjörn Åhs) and to my colleagues in the DReaM group, in particular Alan Smaill and Alan Bundy, for their continuing interest in and support of my work. Thanks also to John Lloyd for help with Gödel.

References

[1] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[2] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, pages 135–162, 1989.

[3] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S.L.H. Clarke, ed., *Proceedings of UK IT 90*, pages 221–6, 1990. Also available from Edinburgh as DAI Research Paper 448.

[4] A. Bundy, A. Smaill, and G. A. Wiggins. The synthesis of logic programs from inductive proofs. In J. Lloyd, editor, *Computational Logic*, pages 135–149. Springer-Verlag, 1990. Esprit Basic Research Series. Also available from Edinburgh as DAI Research Paper 501.

[5] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, 1991. To appear in Artificial Intelligence.

[6] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[7] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[8] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[9] R.L. Constable. Programs as proofs. Technical Report TR 82-532, Dept. of Computer Science, Cornell University, November 1982.

[10] L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In *Proceedings of Eighth International Conference on Logic Programming*, pages 685 – 699. MIT Press, June 1990.

[11] G. Gentzen. *The Collected Papers of Gerhard Gentzen.* North Holland, 1969. edited by Szabo, M.E.

[12] J.T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* PhD thesis, University of Edinburgh, 1991.

[13] P. Hill and J. Lloyd. The Gödel Report. Technical Report TR-91-02, Department of Computer Science, University of Bristol, March 1991. Revised in September 1991.

[14] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[15] R. Kowalski. Algorithm = Logic +Control. *Comm. ACM*, 22:424–436, 1979.

[16] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. North Holland, Amsterdam. 1982.

[17] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[18] C. Phillips. Well-founded induction and program synthesis using proof plans. Research Paper 559, Dept. of Artificial Intelligence, Edinburgh, November 1991.

[19] M. Proietti and A. Pettorossi. Construction of efficient logic programs by loop absorption and generalization. In *Proceedings of Second International Workshop on Meta-Programming in Logic*, April 1990.

[20] G. A. Wiggins. The improvement of prolog program efficiency by compiling control: A proof-theoretic view. In *Proceedings of the Second International Workshop on Meta-programming in Logic*, Leuven, Belgium, April 1990. Also available from Edinburgh as DAI Research Paper No. 455.

[21] G. A. Wiggins. Negation and control in automatically generated logic programs. In A. Pettorossi, editor, *Proceedings of META-92*, 1992.

[22] G. A. Wiggins, A. Bundy, H. C. Kraan, and J. Hesketh. Synthesis and transformation of logic programs through constructive, inductive proof. In K-K. Lau and T. Clement, editors, *Proceedings of LoPSTr-91*, pages 27–45. Springer Verlag, 1991. Workshops in Computing Series.